LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-664263

A Flexible Data Model to Support Multi-Domain Performance Analysis

M. Schulz, A. Bhatele, D. Boehme, P. Bremer, T. Gamblin, A. Gimenez, K. Isaacs

November 13, 2014

8th International Parallel Tools Workshop
Stuttgart, Germany
October 1, 2014 through October 2, 2014

Disclaimer

A Flexible Data Model to Support Multi-Domain Performance Analysis

Martin Schulz[1], Abhinav Bhatele[1], David Böhme[1], Peer-Timo Bremer[1], Todd Gamblin[1], Alfredo Gimenez[1,2], Kate Isaacs[1,2]

[1] Lawrence Livermore National Laboratory, Livermore, CA, USA
[2] University of California at Davis, Davis, CA, USA

Abstract. Performance data can be complex and potentially high dimensional. Further, it can be collected in multiple, independent domains. For example, one can measure code segments, hardware components, data structures, or an application's communication structure. Performance analysis and visualization tools require access to this data in an easy way and must be able to specify relationships and mappings between these domains in order to provide users with intuitive, actionable performance analysis results.
In this paper, we describe a data model that can represent such complex performance data and we discuss how this model helps us to specify mappings between domains. We then apply this model to several use cases both for data acquisition and how it can be mapped into the model, and for performance analysis and how it can be used to gain insight into an application's performance.

1 Motivation

High Performance Computing (HPC) application developers are facing increasing complexity in supercomputer architectures as well as increasing complexity in the simulations that run on them. Modern HPC machines have deep memory hierarchies, complex network topologies, and accelerators such as GPUs and many-core chips. Applications use adaptively refined domain decompositions [11,3,6] and require complex coupling between disparate physics for scale-bridging. To understand these complexities and their interactions, developers must rely on tools for detailed performance information, or they will not be able to optimize their codes. Moreover, measurements must be presented clearly and intuitively to enable actionable insights.

Performance tools have a bad track record in this respect. While they can collect a wide range of performance data and do so efficiently, the data reported by the tools is often very low-level and demands detailed system knowledge from the developer. Some tools, like Scalasca [23] or PerfExpert [5], try to close this gap by presenting higher level analysis results, but such tools are often limited by prior knowledge about potential bottlenecks that had to be coded into the tools; the detection of new bottlenecks or performance problems is often not possible, since it requires an in-depth understanding of code properties and performance data.

One fundamental problem with current tools is that data are displayed in a manner closely related to the way they were measured. While this is straightforward from a tool perspective, it often does not match the intuition of the developer and is hard to

understand. For example, counts of cache misses or branch mispredictions are collected using hardware counters, then displayed on a per-core basis or, at best, mapped to source code. Their relationship to application physics, however, is left for users to infer.

We are addressing this issue with a data model that allows tool developers to a) abstract measurements as values in independent data domains, b) define mappings between domains to describe data transformations and c) use data mapped into more intuitive domains for the visualization of performance information. We build on top of our previous work that proposed a simplified and limited three domain model [16] and we base our model on discussions at a recent Dagstuhl seminar, which discussed the fundamentals of performance visualization [4]. Combined, this enables us to build a new generation of tools that provides developers with intuitive performance visualizations and allows them to gain insight into the performance of complex systems.

We are currently developing an architecture for multi-domain performance analysis, from data acquisition that is capable of gathering data from the entire software stack (including application level information), to data storage and queries, to novel visualization tools that utilize this information and are able to make use of the multi-domain nature of the data. In this paper we present three case studies: the use of hardware to simulation domain mappings, a tool to enable topological views of network data, and a tool to track data movements on NUMA systems. In all cases, the ability to map data from one domain into another for analysis and visualization was instrumental in extracting and understanding the insights necessary for performance optimization.

The remainder of this paper is organized as follows: Section 2 discusses how current tools collect and visualize data and how in some cases this does not match the intuition the user would like to see. We then briefly discuss our previous model [16] and its shortcomings. Based on these observations, we formulate the basics of a generalized model in Section 3 and then introduce an architecture to implement this model on large scale systems in Section 4. In Section 5 we present three case studies showing how the concept of cross-domain analysis can help in detecting performance problems, before we conclude in Section 6.

2 Tools and Their Data Domains

Performance analysis is a well established area in High Performance Computing (HPC) and many tools have been built and are in active use on HPC systems. Examples include Open|SpeedShop [17] and TAU [18], two general performance analysis frameworks; HPCToolkit [14], specializing in sampling based performance analysis; Vampir [15] and Jumpshot [24] for the analysis of communication traces; or mpiP [22] and ompP [9], two profilers for MPI and OpenMP communication respectively. These tools enable users to collect a wide variety of performance measurements based on timing information or using hardware performance counters exposing execution characteristics in the underlying system. In all cases, performance measurements are either tied to the physical hardware they were collected in or to execution objects, such as processes or threads, as defined by the programming model. We refer to this as the *measurement domain*.

While the information available is vast, it is also very low-level and requires substantial effort by a performance analyst to interpret the data and to convert the information into actionable insights. This has lead to the development of tools that aim at automating performance analysis by detecting common bottleneck conditions, rather than displaying raw performance information. The APART project [8] was among the first to formally define a catalogue of bottlenecks. Tools like Scalasca [23] or PerfExpert [5] use such information to search and identify predefined bottleneck patterns.

This approach, though, is naturally limited to existing, well-understood bottlenecks. While this is an important class of problems, new problems, including those caused by emerging architectures and applications, cannot be detected. Those will always require some manual analysis and will have to be done in close collaboration between a performance analyst and the actual application developer.

2.1 Collection vs. Analysis/Visualization Domain

For the latter group, the application developers, though, the information is not represented in an intuitive manner. One of the main reasons for this is that most tools deliver performance analysis results in the domains they were collected in. For example, hardware performance counters are shown per core ID or MPI message traffic is shown per MPI rank. Both are arbitrary spaces that have no intuitive meaning for the developer when taken alone. Developers are more familiar with different domains, like the application domain, e.g., the 2D or 3D domain of a physical phenomena that is being simulated, or the communication structure implemented on top of the MPI rank space.

While we cannot directly measure performance data in these intuitive domains, we can map data from a measurement domain into a (potentially) different visualization or analysis domain [3]. This decouples these two domains: data can be measured in arbitrary domains where it is available, while the user picks the, possibly different, domain in which she wants to see the data. This provides tools with a new flexibility that gives developers a novel way to interpret the data and thereby characterize the performance of their applications.

2.2 The HAC Model

The HAC (Hardware–Application–Communication) model, which we introduced in our previous work [16], was a first attempt to characterize and abstract this concept of decoupling the collection from the visualization domain. This model defined three domains, which we saw as the three most important domains, as well as mappings between them. This is illustrated in Figure 1. The three domains were: H — the hardware domain, which can be used to represent the physical elements of a machine, like cores or network links, and which is used to measure hardware-related data like cache misses, floating point operations or number of packet sent over a link; A — the application domain in which the application's data resides, e.g., the simulated structure in a numerical simulation; and C — the communication domain, which is used to abstract the communication between processes or threads within an application.

[3] We will use these two terms interchangeably in the remainder of the paper.

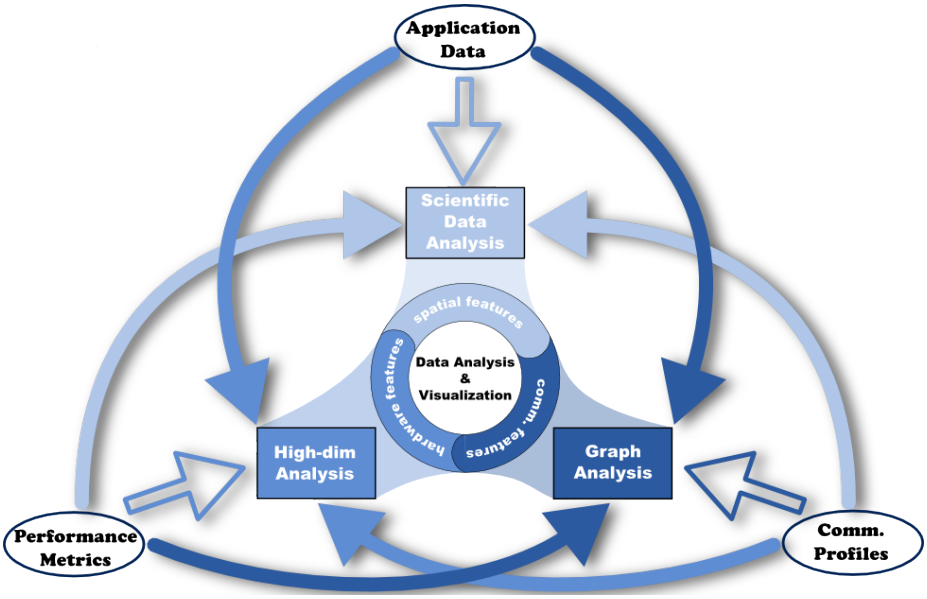


Figure 1. Domains and Mappings in the HAC model.

Each of these domains has its own properties and is associated with its own analysis and visualization techniques. Mappings between the domains can therefore help widen the number of analysis and visualization techniques on the data collected in any of the three domains and can make data collected in two different domains comparable. A straightforward example, which we will discuss in more detail in Section 5.1, is a mapping from the hardware to the application domain that can help attribute performance data collected on hardware resources to the sections of simulation data, which are computed by those resources.

2.3 Missing Elements in the HAC Model

While the HAC model provides the intended abstractions and enables us to provide users with the intended new insights, it only included three very specific data domains. Information on data structures and memory distributions is not included, and the source code and time domains are handled separately and are not part of the model. Reasoning about them is a special case, limiting the scope of possible analyses.

Further, we treated mappings as static properties, which only holds for simple, non-adaptive cases. Codes with dynamic data and execution, such as Adaptive Mesh Refinement codes (AMR) [11,3,6], or dynamic environments, e.g., with thread migration among hardware threads or process migration for load balance optimization, however, need mappings that can be updated based on runtime events. This requires the online collection and integration of meta data.

3 A Generalized Data Model for Performance Tools

We present a generalization of the HAC model that overcomes the limitation described above. This model was the result of discussions at a Dagstuhl seminar on Performance Visualization [4] and has input from both the performance analysis and visualization communities. Figure 2 provides a high-level sketch of the concepts explained below.

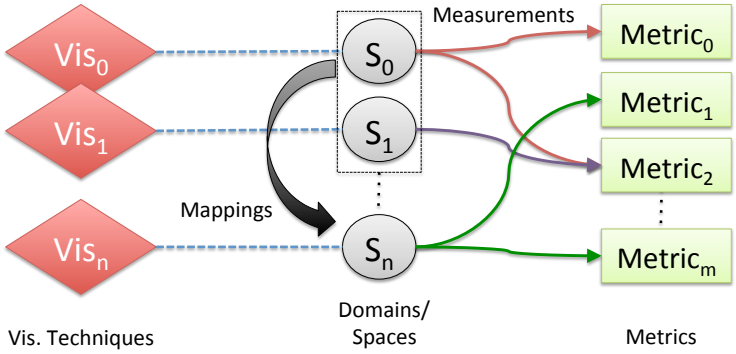


Figure 2. A Generic Data Model to Capture Relationships Between Domains.

3.1 Spaces and Domains

At the core of the abstraction are a set of spaces. Each space is represented by a finite set of tuples and has a crossproduct of types associated with it, such that each type describes one element of the tuple. The number of spaces is not limited. Time and code (represented by calling context trees [20]) are treated the same as any other space. A domain is represented by a crossproduct of spaces.

3.2 Metrics

Metrics are units for individual data points. Examples are floating point operations and MPI message counts. Metrics are typically represented by infinite sets, as not to restrict what can be measured, but may in individual cases be a finite set of possible outcomes.

3.3 Measurements

Measurements capture the data acquisition in performance tools. They are represented as mappings of a crossproduct of spaces, the domain the performance data is collected in, to a metric, the set of possible values for this measurement. To make reasoning about measurements easier, we define a measurement as a unique mapping or function, i.e., for each element of the measurement domain the measurement only maps to at most one element in the metric set. If this is not the case for an experiment, e.g., in profiling tools

that provide multiple data points for each element of a space over time, the domain needs to be modified to allow for this uniqueness, in the example by adding a space representing real or virtual time to the crossproduct that forms the domain.

3.4 Mappings

A mapping, in the sense of the data model, maps one or more spaces (the origin domain) to one or more spaces (the target domain). This allows measurements represented in the target domain to be used in analysis operations on the origin domain. In general, we can distinguish three types of mappings, which are also illustrated in Figure 3:

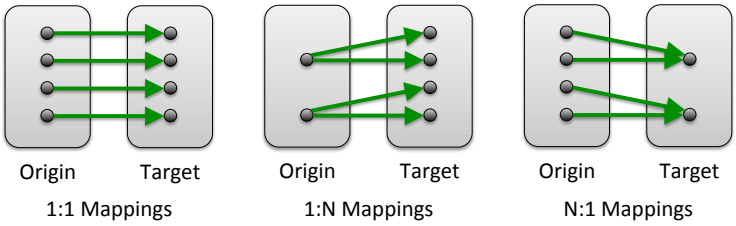


Figure 3. Types of Mappings.

- **1:1 Mappings:** each element of the origin domain is mapped to exactly one element of the target domain. An example of such a 1:1 mapping is the mapping between node coordinates in a network to node IDs, since both domains describe the same physical entity, but using different names or numbering schemes. 1:1 mappings allow a direct translation of measurements in one domain to another.
- **1:N Mappings:** each element of the origin domain is mapped to one or more elements of the target domain. An example for a such a 1:N mapping is the mapping from node IDs in a system to process or MPI rank, since multiple ranks can be on each node. When mapping measurements using a 1:N mapping, measurements from all elements in target domain that map to a single element in the origin domain have to be combined using an aggregation operation. This can be as simple as a sum or average, but can also be a more complex operation such as clustering or statistical outlier detection.
- **N:1 Mappings:** each element of the origin domain is mapped to at most one, not necessarily unique, element of the target domain. An example for a such a N:1 mapping is the mapping of MPI ranks to nodes in a system, since multiple ranks can be on each node. When mapping measurements using a N:1 mapping, a measurement from an element in the target domain must be distributed or spread over all elements in the origin domain that map to it. The semantics of this operation depends on the semantics of the domains. For example, the same measured value could be attributed to each element in the origin domain in full, or the value could be split up based on a distribution function.

Mappings can further be combined into new mappings, allowing a translation over multiple domains from an origin to a target domain. This could also lead to situations in which multiple translations between two domains using different compositions, i.e., a different route through the set of available domains, are possible. Note, though, that not all combined mappings between the same domains carry the same semantics. For example, a 1:1 mapping between two domains may also be representable by a combination of a N:1 and a 1:N mapping, but the latter would include a loss of information by first aggregating measured values before spreading them out again. Choosing the right combination of mappings based on the intended analysis is therefore crucial.

4 An Architecture to Enable Cross-Domain Analysis

To implement this model, we require a performance analysis pipeline that allows us to not only collect performance data, but also collect the necessary context to establish the mappings between spaces. Both should then be stored in a scalable data store that offers a flexible query API to tools so they can extract the data based on the intended visualization domain. We are currently developing such an infrastructure and Figure 4 shows its high-level architecture.

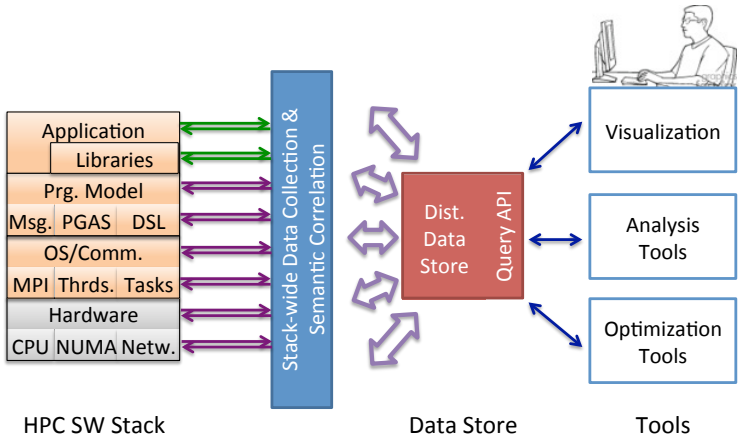


Figure 4. An Architecture for Multi Domain Performance Analysis.

4.1 Software Stack Instrumentation

To extract the necessary context that allows us to establish mappings between spaces, we need access to information from the entire software stack, including from the OS and adaptation decisions it makes, and from runtime systems and their abstractions established for application programmers. For this purpose we need access to the necessary

interfaces, such as the `/proc/` interface in Linux, or access to machine specific instructions and APIs providing us with core and node information. On the programming model side we need standardized mechanism to bridge the abstraction provided by the model. For MPI, the standardized interfaces PMPI (the MPI Profiling Interface) and MPI_T, the newly defined MPI tool information interface [2], can provide the necessary information, but other programming models and runtime systems do not provide the necessary information through a standardized interface.

Over the last year, though, the tools working group in the OpenMP language committee has been developing a new interface that will allow tools to interface with any OpenMP runtime and export the information necessary. This interface, OMPT [7], provides a series of routines to extract runtime information, e.g., to cleanly assemble callstacks, and to insert hooks for events of interest, such as the start of parallel regions or tasks. Initial prototype implementations of OMPT are available on BlueGene/Q machines and Intel platforms. We are currently integrating OMPT into our tools.

4.2 Creating Context

In many cases, though, information provided transparently by runtimes is not sufficient to provide all meta data needed to establish mappings. Many kinds of information are application specific and we need application semantics to properly represent, capture, and store them. Examples are program phases, associations of tasks and data structures, or application specific properties of an input deck. We therefore need an API that enables developers to expose this information in an unobtrusive and tool-independent way.

For this purpose, we have developed a context recording library that gives developers simple commands to annotate application source code and provide context information through key/value pairs. The library then records this context information and makes it available to performance data collection tools. The latter is done by providing a reference to a context information structure, which allows the context recording library to maintain a highly efficient internal representation of the data. At program termination, the library then writes this context information to disk, which later can be extracted during the analysis phase.

Figure 5 shows a few code snippets illustrating how context annotation calls are added to the application source code. In this example, one context key/value pair consists of key `iteration`, which expresses the current iteration that the execution of the target code is in. Developers can also define hierarchical annotations, as shown with the `phase` annotation in the example.

4.3 A Distributed Data Store with Query Access

Once collected, data and metadata will be stored in a large and distributed data store to make it available to tools, enabling queries across different data sources, runs, systems and applications. Our initial implementation used for Boxfish [13] relies on simple, self-describing text files, which capture information from multiple domains with a column per measured metric. This data format has generally been very effective in order to achieve the necessary cross-space connections, but naturally has its limits in terms of

```
Annotation("phase").begin("main"); // phase="main"

Annotation("phase").begin("loop"); // phase="main/loop"
while ( i < count ) {
  Annotation("iteration").set(i);
  do_something( i );
}
Annotation("phase").end(); // ends "main/loop"
Write_results();
Annotation("phase").end(); // ends "main"
```

Figure 5. Example of source code context annotations

scalability and performance. We are therefore currently analyzing both SQL and non-SQL based stores as possible alternatives.

One key aspect of the data store will be that it can execute complex queries, which requires it to track existing mappings and apply them in a suitable order to achieve the necessary transformations. The latter includes finding the right combination of mappings as well as the scheduling of necessary aggregation and spreading functions. If multiple chains of mapping between targets and sources are available and are feasible, the query API should first select a set of likely paths, e.g., ones without huge amounts of data loss due to combining N:1 and 1:N mappings, and then, if ambiguity persists, should forward the decision to the user.

4.4 Analysis and Visualization Tools

From this data store, tools can query performance data based on the visualization domain they prefer or need for their analysis. Generally speaking, we envision three types of tools: a) performance visualization tools, such as Ravel [12], Boxfish [13] and communication visualizations as the one by Sigovan et al. [19], b) analysis tools that preprocess the data and only deliver reduced performance analysis results, or c) tools that can autonomously use the information for tuning, such as Active Harmony [21].

5 Case Studies

In the following we show the advantages and capabilities of our multi-domain approach using three case studies.

5.1 Hardware to Simulation Domain Mappings

To demonstrate how our new abstract data model described above can be instantiated, we first show a straightforward example from a CfD simulation at LLNL, which we previously analyzing using the HAC model [16], and show how it can be represented

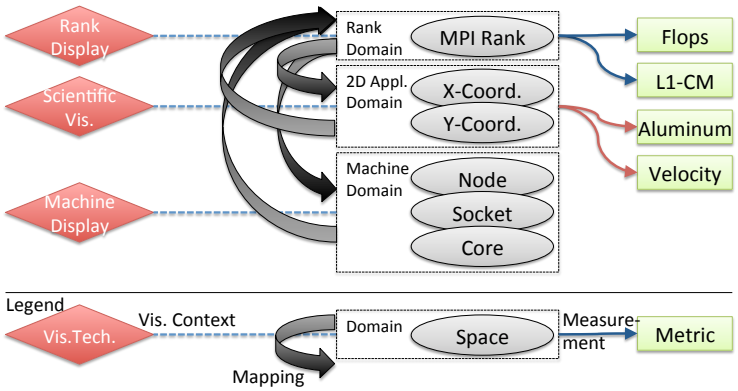
in the new, more general model. In this experiment, we simulate an ablator driving a shock into an aluminum section containing a void, producing a jet of aluminum. The simulation results are from a 12 hour run using a 2D grid running on 256 (8x32) cores of a Linux based cluster with nodes consisting of four Quad-Core AMD processors each. The interconnecting network is Infiniband and we use the MVAPICH MPI implementation. Figures 6(b) and 6(c) show the results of the simulation using nine selected timesteps in regular intervals sorted from left to right. The figures clearly show the aluminum jet on the left side as well as the created shockwave traveling from top to bottom.

Figure 6(a) shows the instantiation of the data model for this experiment: in total we have three domains: the 2D application domain (defined by two spaces for X and Y coordinates) representing the physical structure being simulated, in this case the aluminum shock wave; the MPI rank domain (defined by a single space holding the rank ID), in which most tools operated; and the hardware domain (defined by three spaces, one for each dimension of the machine structure), describing the structure of the machine and its node architecture. During the experiment we collect metrics both in MPI rank space (number of floating point operations and number of L1 cache misses) and two physical properties of the shockwave (the material density itself and the material velocity).

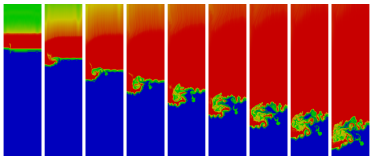
Each of the three domains has it's own visualization technique associated with it: the MPI rank space is represented by a simple bar graph (Figure 6(f) shows the number of floating point operations natively collected in this domain), the application space is shown using a 2D scientific visualization (Figures 6(b) and 6(c) show the two metrics natively collected in this domain); and a representation of the hardware domain with nodes, sockets, and cores is shown in Figure 6(g). Just looking at these native representations, the performance data collected in MPI rank space cannot easily be correlated with any other measurement, which makes it hard to interpret the variations in floating point operations that we observe.

Using mappings between the domains we can establish such a correlation: based on our machine and MPI setup, we can assume a fixed mapping of MPI ranks to cores and therefore we can map the MPI rank domain into the hardware domain (and vice versa) using a static 1:1 mapping. We can further map the rank domain into the application domain by following the domain decomposition: an MPI rank is mapped to all elements in the application domain it is responsible for computing. In this case, the domain decomposition splits up a dense matrix in a 8x32 grid, making each MPI rank responsible for one of 256 sub-squares of the full problem. Since there are more data points in the application domain, the mapping from MPI rank domain to the application is a 1:N mapping, while the reverse direction is a N:1 mapping.

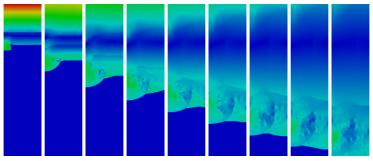
Figures 6(d) and 6(e) show the two metrics measured in the MPI rank domain mapped into the 2D application domain and shown using the matching visualization technique. Since multiple elements in the application domain map to a single MPI rank, the associated N:1 mapping from the application domain to the rank domain spreads the measurements of a single rank to all elements in application domain, resulting in a more coarse grained representation. Nevertheless, this displays shows a clear correlation with



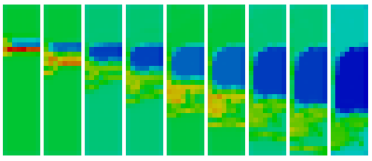
(a) Abstract representation using the generalized data model



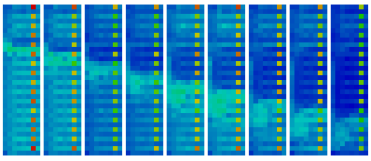
(b) Aluminum in 2D Scientific Vis.



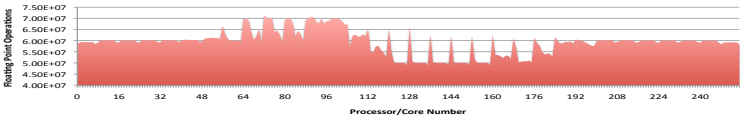
(c) Velocity in 2D Scientific Vis.



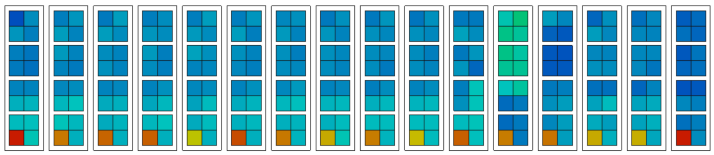
(d) Flop/s in 2D Scientific Vis.



(e) Cache Misses in 2D Scientific Vis.



(f) Flop/s, measured per rank, displayed per rank



(g) Cache Misses display in machine topology

Figure 6. Example instantiation of the data model using a CfD code.

the simulated physics in Figures 6(b) and 6(c) for both metrics, which wasn't obvious before in the MPI rank display.

The visualization of the L1 cache misses in Figure 6(e) shows a second phenomenon: a series of ranks with high numbers of cache misses represented by dots on the right side of the figure for each timestep, which overlays the physics, but does not seem to be correlated. When we map the same data into the hardware domain (Figure 6(g)) we see that each of these cores maps to the same core ID on each node and we tracked this phenomena to the MPI library, which uses this core of aggregating node local information for collective operations.

Using the two different mappings into two domains enabled us to distinguished these two separate phenomena and what they correlate to. It also showed that they need to be analyzed separately since they have two different sources: one related to the physics being simulated, i.e., input dependent, and one related to the base architecture, i.e., input independent.

5.2 Domain Mappings in Boxfish

In order to automate the analysis process described above, we have implemented a prototype of the query and mapping functionality in the Boxfish toolset [13]. An annotated screenshot of our tools with example mappings is shown in Figure 7.

The tool offers a range of visualization techniques, shown in the bottom left of the figure. Our current implementation focuses mainly on visualization of communication data, but it can be extended with plugins to other domains. The input data and its metrics are shown in the top left window in the figure. Each *table* represents one measurement domain and the elements below it show the various metrics available in that domain.

A user can select any visualization domain by dragging it into the main window. The example shows a 3D visualization of a torus network, as used in Blue Gene/L and P as well as Cray HPC systems. Once the display is activated, the user can select any number of metrics available as input data and attempt a mapping of this data onto the visualization domain, again by simply dragging the metric onto the display. At that time, Boxfish searches all available mappings, provided through plugins into the tool, to see if a mapping from the selected visualization to the selected input metrics and domain can be established. If the mapping is unique, the data is simply displayed accordingly. If multiple mappings are available, though, the tool presents this choice to the user. In the example in the figure, the user drags a node metric onto the torus display. However, since the tool also knows how to map node IDs to link IDs (since this is how link data is measured on Blue Gene systems), Boxfish offers the user the choice of visualizing the data on nodes or links. The user picks the intended one by dragging the data onto that choice.

While the choice is clear to a human user in this case, the system cannot distinguish between these two choices without further semantic information. This shows the generality of the approach, but also a shortcoming. We will investigate how to add semantic information to reduce choices offered by Boxfish to reasonable mappings in future work. In other examples, though, several choices do make sense (e.g., attributing packets sent over a link can be to a link to show network traffic vs. to a node to show

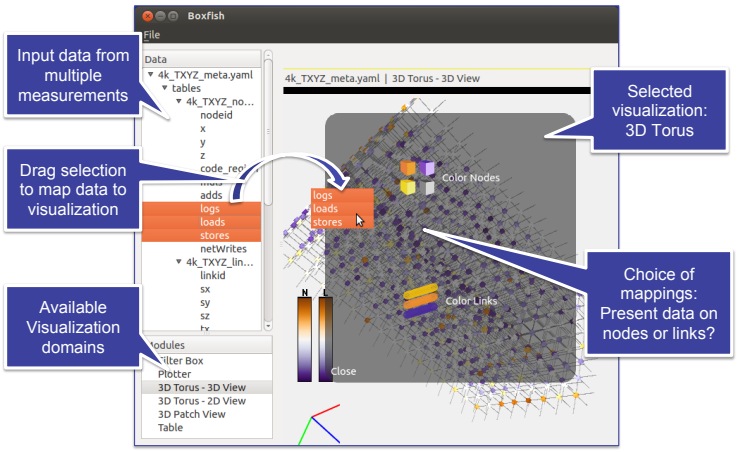


Figure 7. Multidomain visualization in Boxfish.

network load on a node). In these cases the system provides users with the necessary flexibility to choose the option suitable for the intended analysis.

5.3 NUMA Optimizations with MemAxes

In addition to Boxfish, we started using the concept of domain mappings in other tools. One example is MemAxes [10], a tool to visualize memory traffic in NUMA nodes. It uses Intel's Precise Event Based Sampling (PEBS) to sample all memory accesses and their attributes, including latency, target address and memory hierarchy they hit. This information is further enhanced with context information containing bowth source code meta data (e.g., which element in which structure an address maps to) and machine information (e.g., core ID to NUMA domain mappings). Figure 8 shows an overview of the graphical user interface. The elements labeled with A show the application view of the collected data, mapping it to data structures and source code. The view labeled B shows a mapping of the collected data to the hardware architecture of the underlying system: HW threads are aligned on the outer most ring, following by the — partially shared — caches all the way to the two NUMA domains in the innermost circle. The amount of communication between the layers is shown through the thickness of the lines between the layers. Users can select only parts of the data being shown, which updates the various views in the GUI. The graph in window labeled with C shows the percentage of the data that is currently shown.

The selection can be done using a parallel coordinates view in the bottom most window D. Each vertical bar (axis) represent one measurement on one metric (across all spaces in which data is collected) and a single sample is drawn as a line between all axes showing the value in that particular measurement for this sample. Users can select arbitrary ranges on any of the shown measurements by marking a region of the axis, which will then color all lines crossing the axis at that measurement red. This

visualization shows how other measurements and spaces correlate to elements being selected in the chosen space.

An example of how this can be used is shown in Figure 9: We ran LULESH [1], a shock-hyrdo proxy application developed at LLNL on a NUMA system with 32 HW threads split into two NUMA domains. The results of the default run are shown in Figure 9(a). Using the tool, we selected all samples representing a continuous range in the code's main data structure (right bar) and observed the corresponding memory traffic. As the architectural view clearly shows, the contiguous mapping of this data structure matches the contiguous set of core IDs, but those are mapped in alternating order to both NUMA nodes (second bar from the right). As a consequence, the system experiences significant traffic between NUMA regions.

To optimize this behavior we rearrange the cores by alternating core IDs, shown in Figure 9(b). As a result, consecutive ranges in the main data structure are now mapped to core IDs in the same way they are in the NUMA domain. This lead to a performance improvement of over 10%.

6 Conclusions

Performance analysis plays a critical role in optimizing codes on current and future platforms. The ever increasing complexity of both system architectures and applications makes this a difficult task and application developers rely on tools to provide them with sufficient insights into the performance behavior of their applications. While many tools exist and can provide a vast amount of data, the resulting information is often low-level and hard to interpret. One reason for this is that information is displayed in domains in which data was collected, but those domains are not necessarily the intuitive ones that help application developers extract actionable insights.

In this work we introduced a generic data model that helps us describe independent data domains for both the collection of measurements and their visualization/analysis. By establishing mappings between domains we can then translate data collected in one domain into another and use this to show performance data in other domains, to make data comparable across domains, and provide intuitive visualizations in domains that application developers are familiar with. This concept has already been helpful in many cases and we showed three of them in this paper: mapping of performance data to the application domain of a CfD simulation helped us correlate performance data with the underlying physics being simulated and to distinguish those effects from a machine related phenomena; the Boxfish tool enabled us to understand network performance data by mapping it to the underlying physical network architecture, in our case a 3D torus; and correlating memory access information from the machine architecture and the application data structures allowed us to identify and correct excessive NUMA accesses for a shock-hydro code.

In summary, the new data model and its ability to map performance data across domains enables us to create a new generation of tools that provide more insights into an application's performance and provide this information in an intuitive way that enables optimizations. It is also flexible enough to carry forward to next generation systems and applications, incl. new programming models and abstractions by integrating more

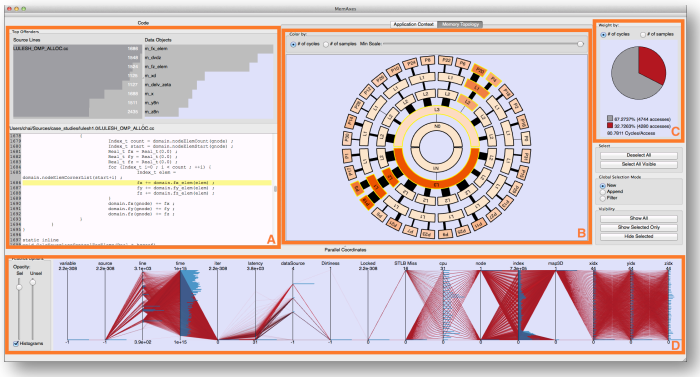
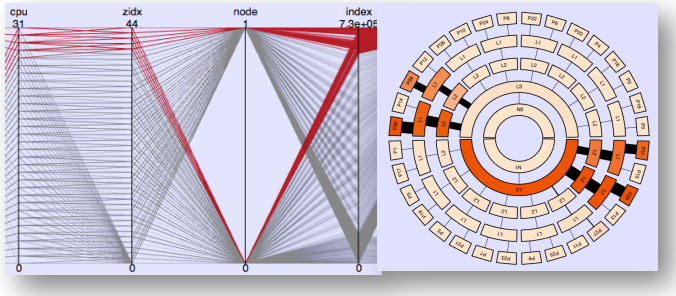
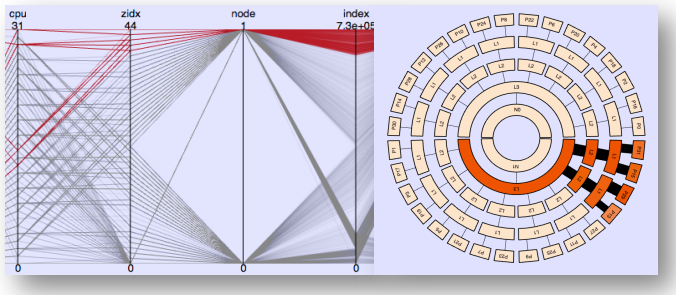


Figure 8. The Memaxis User Interface.



(a) Initial version of LULESH: memory accesses across NUMA domains.



(b) LULESH after memory access optimizations.

Figure 9. Using MemAxes to expose memory traffic in NUMA systems.

diverse domains and measurement techniques. Equally important, it provides a way to formalize performance data to enable a closer interaction between the performance analysis community on one side and the data visualization and analysis communities on the other, which will allows for an easier transfer of tools and techniques.

Acknowledgement

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 and supported by Office of Science, Office of Advanced Scientific Computing Research as well as the Advanced Simulation and Computing (ASC) program.

References

1. Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory. Technical Report LLNL-TR-490254.
2. MPI Standard 3.0. http://www.mpi-forum.org/docs/docs.html.
3. J. Bell, A. Almgren, V. Beckner, M. Day, M. Lijewski, A. Nonaka, and W. Zhang. BoxLib User?s Guide. `https://ccse.lbl.gov/BoxLib/BoxLibUsersGuide.pdf`, May 2013.
4. Peer-Timo Bremer, Bernd Mohr, Valerio Pascucci, and Martin Schulz. Connecting Performance Analysis and Visualization to Advance Extreme Scale Computing, January 2014. `http://www.dagstuhl.de/de/programm/kalender/semhp/?semnr=14022`.
5. Martin Burtscher, Byoung-Do Kim, Jeff Diamond, John McCalpin, Lars Koesterke, and James Browne. Perfexpert: An easy-to-use performance diagnosis tool for hpc applications. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.
6. P. Colella, D. T. Graves, N. D. Keen, T. J. Ligocki, D. F. Martin, P. W. McCorquodale, D. Modiano, P. O. Schwartz, T. D. Sternberg, and B. Van Straalen. Chombo software package for amr applications design document. Technical report, Applied Numerical Algorithms Group, Computational Research Division, Lawrence Berkeley National Laboratory, Berkeley, CA, April 15, 2009 2009.
7. Alexandre E. Eichenberger, John Mellor-Crummey, Martin Schulz, Michael Wong, Nawal Copty, Robert Dietrich, Xu Liu, Eugene Loh, and Daniel Lorenz. OMPT: An OpenMP tools application programming interface for performance analysis. In Alistair P. Rendell, Barbara M. Chapman, and Matthias S. Müller, editors, *OpenMP in the Era of Low Power Devices and Accelerators*, number 8122 in Lecture Notes in Computer Science, pages 171–185. Springer Berlin Heidelberg, January 2013.
8. T. Fahringer, M. Gerndt, G. Riley, and J.L. Traff. The APART Specification Language (illustrated with MPI). Technical Report FZJ-ZAM-IB-2001-08, Forschungszentrum Jülich, 2001.
9. Karl Fürlinger and Michael Gerndt. ompP: A profiling tool for OpenMP. In *Proceedings of the First International Workshop on OpenMP (IWOMP 2005)*, Eugene, OR, June 1-4 2005.
10. Alfredo Giménez, Todd Gamblin, Barry Rountree, Abhinav Bhatele, Ilir Jusufi, Peer-Timo Bremer, and Bernd Hamann. Dissecting on-node memory access performance: A semantic

approach. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14. IEEE Computer Society, November 2014. LLNL-CONF-658626.

11. Richard D. Hornung and Scott R. Kohn. Managing application complexity in the samrai object-oriented framework. *Concurrency and Computation: Practice and Experience*, 14(5):347–368, 2002.
12. Katherine Isaacs, Peer-Timo Bremer, Ilir Jusufi, Todd Gamblin, Abhinav Bhatele, Martin Schulz, and Bernd Hamann. Combing the Communication Hairball: Visualizing Large-Scale Parallel Execution Traces using Logical Time. In *Proceedings of IEEE InfoVis*, InfoVis '14, November 2014.
13. Aaditya G. Landge, Joshua A. Levine, Katherine E. Isaacs, Abhinav Bhatele, Todd Gamblin, Martin Schulz, Steve H. Langer, Peer-Timo Bremer, and Valerio Pascucci. Visualizing network traffic to understand the performance of massively parallel simulations. In *IEEE Symposium on Information Visualization (INFOVIS'12)*, Seattle, WA, October 14-19 2012. LLNL-CONF-543359.
14. J. Mellor-Crummey, R. Fowler, and G. Marin. HPCView: A tool for top-down analysis of node performance. *The Journal of Supercomputing*, 23:81–101, 2002.
15. W. E. Nagel, A. Arnold, M. Weber, H. C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer*, 12(1):69–80, 1996.
16. M. Schulz, J.A. Levine, P.-T. Bremer, T. Gamblin, and V. Pascucci. Interpreting performance data across intuitive domains. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 206–215, September 2011.
17. Martin Schulz, Jim Galarowicz, Don Maghrak, William Hachfeld, David Montoya, and Scott Cranford. Open|speedshop: An open source infrastructure for parallel performance analysis. *Scientific Programming*, 16(2-3):105–121, 2008.
18. S. Shende and A. D. Malony. The tau parallel performance system. *International Journal of High Performance Computing Applications, ACTS Collection Special Issue*, 2005.
19. Carmen Sigovan, Chris W. Muelder, and Kwan-Liu Ma. Visualizing large-scale parallel communication traces using a particle animation technique. *Computer Graphics Forum*, 32(3):141–150, June 2013.
20. Nathan R. Tallent, Laksono Adhianto, and John M. Mellor-Crummey. Scalable identification of load imbalance in parallel executions using call path profiles. In *Proceedings of IEEE/ACM Supercomputing '10*, November 2010.
21. Cristian Tapus, I-Hsin Chung, and Jeffrey K. Hollingsworth. Active harmony: Towards automated performance tuning. In *In Proceedings from the Conference on High Performance Networking and Computing*, pages 1–11, 2003.
22. Jeffrey Vetter and Chris Chambreau. mpiP: Lightweight, Scalable MPI Profiling. `http://mpip.sourceforge.net`.
23. Felix Wolf, Brian Wylie, Erika Abraham, Daniel Becker, Wolfgrang Frings, Karl Fuerlinger, Markus Geimer, Marc-Andre Hermanns, Bernd Mohr, Shirley Moore, and Zoltan Szebenyi. Usage of the SCALASCA Toolset for Scalable Performance Analysis of Large-Scale Parallel Applications. In *Proceedings of the 2nd HLRS Parallel Tools Workshop*, Stuttgart, Germany, july 2008.
24. O. Zaki, E. Lusk, W. Gropp, and D. Swider. Toward Scalable Performance Visualization with Jumpshot. *International Journal of High Performance Computing Applications*, 13(3):277–288, 1999.